The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Fast 3D Wavelet Transform on Multicore and Manycore Computing Platforms

V. Galiano · O. López-Granado · M.P. Malumbres · H. Migallón

Received: date / Accepted: date

Abstract Three-dimensional wavelet transform (3D-DWT) has focused the attention of the research community, most of all in areas such as video watermarking, compression of volumetric medical data, multispectral image coding, 3D model coding and video coding. In this work, we present several strategies to speed-up the 3D-DWT computation through multicore processing. An in depth analysis about the available compiler optimizations is also presented. Depending on both the multicore platform and the GOP size, the developed parallel algorithm obtains efficiencies above 95% using up to four cores (or processes), and above 83% using up to twelve cores. Furthermore, the extra memory requirements is under 0.12% for low resolution video frames, and under 0.017% for high resolution video frames. In this work, we also present a CUDA based algorithm to compute the 3D-DWT using the shared memory for the extra memory demands, obtaining speed-ups up to 12.68 on the manycore GTX280 platform. In areas such as video processing or ultra high definition image processing, the memory requirements can significantly de-

This research was partially supported by the Spanish Ministry of Science and Innovation under grant numbers TIN2011-27543-C03-03, TIN2011-26254 and TIN2010-12011-E.

V. Galiano
Physics and Computer Architecture Dept. Miguel Hernández University, 03202 Elche, Spain.

O. López-Granado
Physics and Computer Architecture Dept. Miguel Hernández University, 03202 Elche, Spain.

M.P. Malumbres
Physics and Computer Architecture Dept. Miguel Hernández University, 03202 Elche, Spain.

H. Migallón
Physics and Computer Architecture Department. Miguel Hernández University, 03202 Elche, Spain.
Tel.: +34-966658390
Fax: +34-966658814
E-mail: hmigallon@umh.es

grade the developed algorithms, however, our algorithm increases the memory requirements in a negligible percentage, being able to perform a nearly in-place computation of the 3D-DWT whereas in other state-of-the-art 3D-DWT algorithms it is quite common to use a different memory space to store the computed wavelet coefficients doubling in this manner the memory requirements.

Keywords 3D wavelet transform · video coding · parallel algorithms · in-place computing · OpenMP · CUDA · GPU

1 Introduction

In the last years, the three-dimensional wavelet transform (3D-DWT) has focused the attention of the research community, most of all in areas such as video watermarking [1] and 3D coding (e.g., compression of volumetric medical data [2] or multispectral images [3], 3D model coding [4], and especially, video coding). 3-D subband video coding is an alternative to the traditional motion-compensated Discrete Cosine Transform (DCT) coding. The 3D subband coding uses the discrete wavelet transform (DWT), which achieves better energy compaction, instead of the DCT.

Podilchuk, et al., utilized 3-D spatio-temporal subband decomposition and geometric vector quantization (GVQ) [5]. Taubman and Zakhor presented a full color video coder based on 3-D subband coding with camera pan compensation [6]. The embedded zerotree wavelet (EZW) algorithm developed by Shapiro [7], the set partitioning in hierarchical trees (SPIHT) algorithm developed by Said and Pearlman [8] and the lower tree wavelet encoder (LTW) developed by Oliver and Malumbres [9] exploit the similarity between different wavelet subbands based on a wavelet tree structure. They provide remarkably good performance on 2-D images, with low computational complexity. Then, an adapted version of an image encoder can be used, taking into account the new dimension. For instance, the two dimensional (2D) embedded zero-tree (IEZW) method has been extended to 3D IEZW for video coding by Chen and Pearlman[10], and showed promise of an effective and computationally simple video coding system without motion compensation, obtaining excellent numerical and visual results. A 3D zero-tree coding through modified EZW has also been used with good results in compression of volumetric images[11]. In [12] and [13], instead of the typical quad-trees of image coding, a tree with eight descendants per coefficient is used to extend both SPIHT and LTW image encoders to 3D video coding.

Several attempts have been made in order to accelerate the DWT, especially the 2D DWT, exploiting both multicore architectures and graphic processing units (GPUs). In [14], a SIMD algorithm runs the 2D-DWT on a GeForce 7800 GTX using Cg and OpenGL, with a remarkable speed-up. A similar effort in [15] combined Cg and the 7800 GTX to report a 1.2x-3.4x speed-up versus a CPU counterpart. In [16], a CUDA implementation for the 2D-FWT runs more than 20 times faster than the sequential C version on a CPU, and it is more than twice as fast as the optimized OpenMP and Pthreads versions implemented on multicore CPUs. In a previous work [17], we presented both multicore and GPU implementations for the 2D-DWT obtaining speed-ups up to 7.1 and 8.9 on a multicore platform using eight and ten processes, respectively when compared to the CPU sequential algorithm.

This work extends our analysis to the 3D-DWT, analyzing the compiler flags impact as well as the different optimizations applied. We analyze the computational behavior in order to set the optimal performance parameters. Furthermore, we compare our results to the ones obtained in a recent 3D-DWT implementation presented in [18].

The rest of the paper is organized as follows. Section 2 presents the foundations of the 3D-DWT. Sections 3 and 4 describe our implementation proposals on multicore and manycores respectively, while in Section 5 we analyze their performance. Finally in Section 6 some conclusions are drawn.

2 Three Dimensional Wavelet Transform

The DWT is a multiresolution decomposition scheme for input digital signals, see detailed description in [19]. The source signal is firstly decomposed into two frequency subbands, low-frequency (low-pass) subband and high-frequency (high-pass) subband. For the classical DWT, the forward decomposition of a signal is implemented by a low-pass digital filter H and a high-pass digital filter G. Both digital filters are derived using the scaling function $\Phi(t)$ and the corresponding wavelet functions at different frequency scales $\Psi(t)$. The system downsamples the signal to half of the filtered results in the decomposition process. If four-tap and non-recursive FIR filters are considered, the transfer functions of H and G can be represented as follows:

$$H(z) = h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} \tag{1}$$

$$G(z) = g_0 + g_1 z^{-1} + g_2 z^{-2} + g_3 z^{-3} \tag{2}$$

To use the wavelet transform for volume and video processing we must implement a 3D version of the analysis and synthesis filter banks. In the 3D case, the 1D analysis filter bank is applied in turn to each of the three dimensions. If the data is of size N_1 by N_2 by N_3, then after applying the 1D analysis filter bank to the first dimension we have two subband data sets, each of size $\frac{N_1}{2}$ by N_2 by N_3. After applying the 1D analysis filter bank to the second dimension we have four subband data sets, each of size $\frac{N_1}{2}$ by $\frac{N_2}{2}$ by N_3. Applying the 1D analysis filter bank to the third dimension gives eight subband data sets, each of size $\frac{N_1}{2}$ by $\frac{N_2}{2}$ by $\frac{N_3}{2}$ (see Fig. 1).

After applying the 3D-DWT on a group of video pictures (GOP), a 2D spatial DWT and a 1D temporal DWT, we obtain eight first level wavelet subbands (typically named as LLL_1, LHL_1, LLH_1, LHH_1, HLL_1, HHL_1, HLH_1, HHH_1). Further wavelet decompositions can be done, focusing on the low-frequency subband (LLL_1), achieving in this way a second-level wavelet decomposition, and so on (see example in Fig. 2).

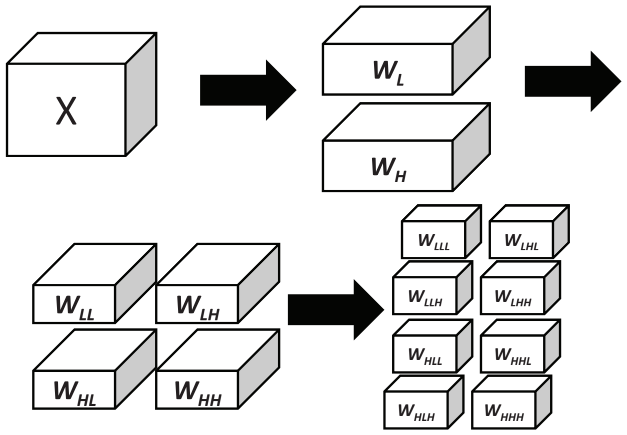


Fig. 1 The resolution of a 3-D signal is reduced in each dimension

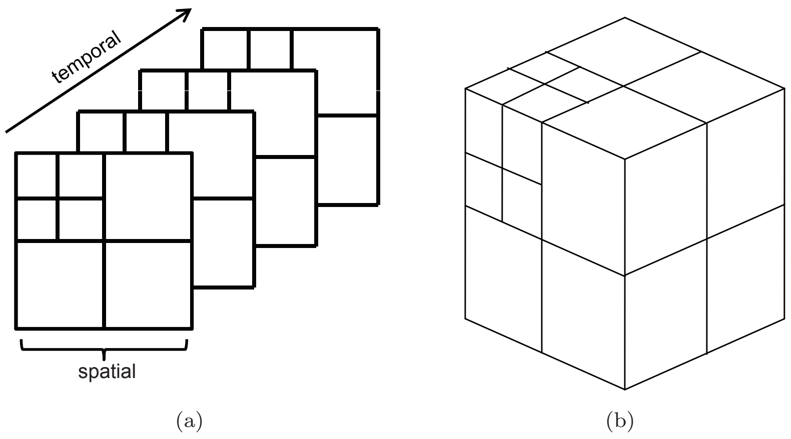


Fig. 2 Overview of the 3D-DWT computation in a two-level decomposition using the regular 3D-DWT algorithm.

In this work we will use the Daubechies 9/7 filter for both the spatial and temporal decompositions. In addition, the regular filter-bank convolution is considered to develop the three-dimensional wavelet transform, based on the results obtained in [17] for the two-dimensional wavelet transform case. In particular, in [17] we obtained best results, in terms of computational times and in terms of parallel performance, applying the regular filter-bank convolution than applying the lifting scheme.

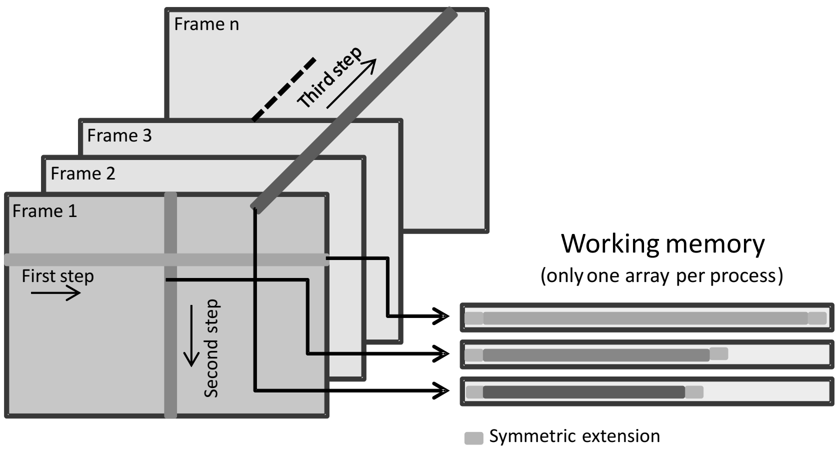


Fig. 3 3D-DWT one level structure computation.

3 Multicore 3D Wavelet Transform

As we have said, the Daubechies 9/7 filter, proposed in [19], has been used to perform the regular filter-bank convolution in order to develop the parallel 3D-DWT algorithm. In [17] we proposed the convolution-based parallel 2D-DWT using an extra memory space in order to perform a nearly in-place computation, avoiding the requirement of twice the image size to store the computed coefficients. This strategy is also followed to develop the parallel 3D-DWT algorithm.

We want to remark that we use four decomposition levels in order to compute the 3D-DWT, and, as we have said in Section 2, the computation of each wavelet decomposition level is divided into two main steps. In the first step the 2D-DWT is applied to each frame of the current GOP, and in the second step the 1D-DWT is performed to consider the temporal axis. We have used the symmetric extension technique in order to avoid the border effects on both the frame borders and the GOP borders.

So, in order to apply the 2D-DWT to each frame we compute the components associated to each row, after that, we compute the components associated to each column. Finally, in the third step, the temporal 1D-DWT, we compute the components associated to the third dimension, the temporal dimension (see Fig. 3). The current working data (row, column or array in temporal axis) are copied into the extra memory reserved to process it over this memory. We perform all computations over this working memory and, after the computation is finished, we store the computed coefficients in the original pixel positions of the source data. As we have said, in order to avoid the border effects, we perform the symmetric extension in the working memory previous to start the computation. As we have seen in Section 2, the resolution of the 3-D signal is reduced in each wavelet decomposition. The computation to obtain each coefficient is an optimized classical convolution filtering.

Frame Size	Processes	Extra memory size (pixels)	Increment (%) GOP: 32
352 x 288	1	360	0.0110
	2	720	0.0221
	4	1440	0.0443
	6	2160	0.0665
	10	3600	**0.1109**
1280 x 640	1	1288	0.0024
	2	2576	0.0049
	4	5152	0.0099
	6	7728	0.0148
	10	12880	0.0247
1920 x 1024	1	1928	0.0016
	2	3856	0.0032
	4	7712	0.0065
	6	11568	0.0098
	10	**19280**	0.0164

Table 1 Amount of extra memory size.

If we consider the first step (i.e. the 2D-DWT applied to each video frame), the extra memory size depends on both, the row size or column size (the largest one), and the number of processes in the parallel algorithm. The extra memory must include the pixels required to perform the symmetric extension, therefore, for the Daubechies 9/7 filter we must extend with four elements on each border. As we can see in Fig. 3, the extra memory stores the working data row, the working data column and the working array of the temporal axis of the first stage of the 2D-DWT, the second stage of the 2D-DWT, and the 1D-DWT in the temporal axis, respectively, plus the pixels required to perform the symmetric extension.

Table 1 shows the extra memory size (in pixels) and the percentage of memory increase for several video frame resolutions and number of processes used in the parallel algorithm. Note that each process stores its own working pixels which are not shared with other processes. The worst case in Table 1, attending at memory increase, is a very small value equal to 0.1109%. If the GOP size is larger than the row or column size, the amount of required extra memory is fixed by the GOP length. For the results in Table 1 the percentage has been obtained considering a GOP size equal to 32.

In the second step of the 3D-DWT (i.e. the temporal 1D-DWT), we perform the symmetric extension in order to avoid the border effects in the temporal domain. In all experiments performed the maximum GOP size considered is 128, therefore the extra memory used in the first step is enough to be reused in the second step.

We have used the OpenMP [20] paradigm in order to develop the parallel 3D-DWT algorithm. The multicore platforms used in our tests are:

– Intel Core 2 Quad Q6600 2.4 GHz, with 4 cores.
– HP Proliant SL390 G7 with two Intel Xeon X5660, each CPU with six cores at 2.8 GHz.

We analyze some OpenMP-based techniques to parallelize the two main steps. The techniques tested to parallelize the 3D-DWT algorithm are:

– Automatic OpenMP parallel loops.
– Parallel sections.
– Load balancing according to the thread rank.

In all experiments performed the parallel sections technique has not obtained good results, then we do not use this technique in the following attempts. Therefore we present some parallel options following the above approximations, the automatic OpenMP parallel loops and the load balancing according to the thread rank. It is important to remark that using both parallel sections technique and automatic OpenMP parallel loops, the extra memory size needed does not depend on the number of parallel processes used. In both cases, all processes work with the same working memory.

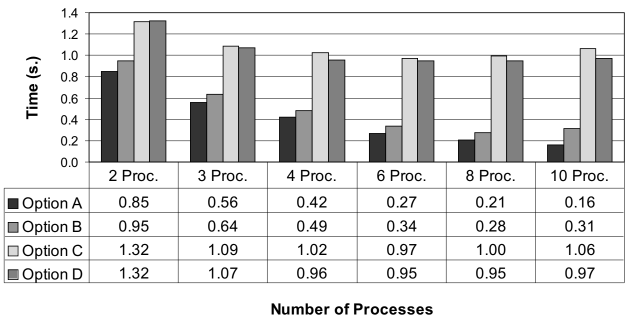
We have tested the following four options:

– Option A: The initial 2D-DWT applied to each video frame is computed by all processes which cooperate in the task, such that a process computes a block of contiguous rows and a block of contiguous columns. The temporal 1D-DWT is computed assigning a block of contiguous columns to each process according to its parallel rank.
– Option B: In the first step only one process computes the initial 2D-DWT of each video frame. The temporal 1D-DWT is computed assigning a block of columns to each process according to its parallel rank.
– Option C: The initial 2D-DWT is computed as in option A. The temporal 1D-DWT is computed using automatic OpenMP parallel loops.
– Option D: The initial 2D-DWT is computed as in option A. The temporal 1D-DWT is computed parallelizing the row procedure instead of the column procedure, that is assigning a block of contiguous rows to each process according to its parallel rank.

In Fig. 4 we present results in order to analyze the parallel strategies described. The best performance, in all experiments performed, is achieved by balancing the computational load according to the thread rank, as we can see in Fig. 4.

Fig. 4 presents results for a grayscale video frame size of 1280×640 pixels and a GOP size equal to 64 on the multicore HP Proliant SL390, using the ICC compiler with *-fast* flag compiler option and varying the number of processes. Note that each core computes only one process. We can observe that option A obtains good efficiencies and scalability, while for the other options, the efficiency degrades as the number of processes increases. The automatic OpenMP loops degrades the performance and the scalability. Note that in this case all processes must use the same extra working memory.

Fig. 5 presents the parallel structure used in the option A in order to balance the computational load. As we have said, in the first stage, each process computes a group of adjacent rows, after that, each process computes a group of adjacent columns, at this point the 2D-DWT transform has been performed.



	2 Proc.	3 Proc.	4 Proc.	6 Proc.	8 Proc.	10 Proc.
■ Option A	0.85	0.56	0.42	0.27	0.21	0.16
▨ Option B	0.95	0.64	0.49	0.34	0.28	0.31
□ Option C	1.32	1.09	1.02	0.97	1.00	1.06
▨ Option D	1.32	1.07	0.96	0.95	0.95	0.97

Fig. 4 Computational times for the multicore 3D-DWT algorithm. Compiler: ICC. Compiler flags: -fast -openmp. Frame size: 1280 × 640. Multicore HP Proliant SL390.

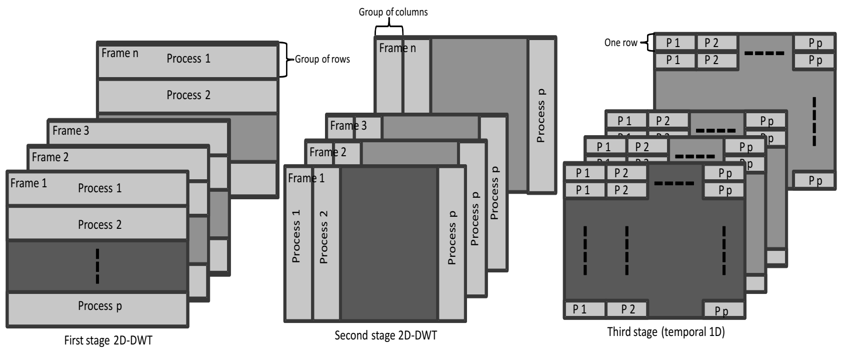
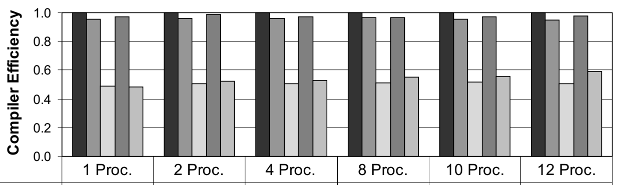


Fig. 5 Parallel structure for the optimal parallel option.

And in the third stage a group of temporal arrays with a group of adjacent columns of each row is assigned to each process. Obviously, as each process computes a different row, column or temporal array, each process works with its own extra memory space increasing the memory requirements as we increase the number of processes in the parallel execution. The optimal choice, the option A, has been used in the rest of the experiments shown in this paper.

We have also analyzed the behavior depending on the compiler and the flags used to build the 3D-DWT algorithm. We have tested the ICC [21] compiler, a corporate tool from Intel, and the GCC [22] compiler, which is a free compiler developed by the GNU project. In the multicore HP Proliant SL390 compilers available are the GCC 4.1.2 and the ICC 12.0.0, however in the multicore Q6600 the only available compiler is the GCC 4.4.3. Fig. 6 shows the compiler efficiency, respect to the best option, for a grayscale video frame size of 1280 × 640 pixels and a GOP size equal to 64 on the multicore HP Proliant SL390. The best option is obtained using the ICC compiler and the *-fast* flag (note that *-fast* is a shorthand that includes the flags *-O3 -ipo -static -xHOST -no-prec-div*). Note that the efficiencies showed in Fig. 6 are computed respect to the computational time obtained using the ICC compiler and the compiler flag *-fast* and the number of processes used in each experiment. Also in Fig. 6 we



	1 Proc.	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ icc -fast	1.00	1.00	1.00	1.00	1.00	1.00
▨ icc -O3	0.96	0.96	0.96	0.96	0.96	0.95
□ icc	0.49	0.50	0.50	0.51	0.52	0.51
▨ gcc -O3	0.97	0.99	0.97	0.96	0.97	0.98
□ gcc	0.48	0.52	0.53	0.55	0.55	0.59

Number of Processes

Fig. 6 Compiler efficiency for the multicore 3D-DWT algorithm. Frame size: 1280 × 640. Multicore HP Proliant SL390.

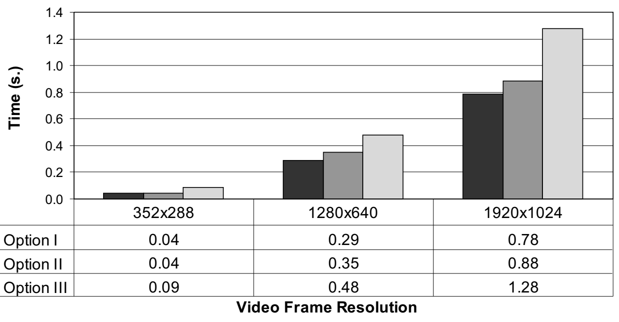
can observe that the performance of both compilers remains unchanged as we increase the number of processes. This conclusion can be applied to the use of the different compiler flags. It is important to remark that both compilers offer the same performance when we use the same optimization flags and the ICC compiler obtains a slight performance increase applying the *-fast* flag .

4 Manycore 3D Wavelet Transform

In order to develop the algorithm presented in Section 3 on a manycore GPU architecture we must consider the GPU model we are going to use. So in the CUDA parallel programming model [23, 24], an application consists of a sequential host program, that may execute parallel programs known as kernels on a manycore platform. Considering CUDA as development tool, a kernel is an Single Program Multiple Data (SPMD) computation that is executed using a large number of parallel threads organized into a set of blocks namely grid.

In order to develop and run the CUDA algorithm we have tested the following three options:

– Option I: The initial 2D-DWT applied to each video frame is computed by two different kernels, the first one performs the horizontal process and the second one performs the vertical process. Each thread computes pixels with stride equal to the number of threads per block on both dimensions. The temporal 1D-DWT is computed only by one kernel that computes the elements corresponding to one column, this kernel is executed the number of rows times.
– Option II: In the initial 2D-DWT each thread computes adjacent pixels on both dimensions, all threads cooperate to compute the coefficients of each frame. The temporal 1D-DWT is computed as in option I.
– Option III: The initial 2D-DWT is computed as in option I. The temporal 1D-DWT is computed by one kernel with a two dimensional grid of blocks, equal to the frame dimensions, such a way that each block computes one element.



	352x288	1280x640	1920x1024
■ Option I	0.04	0.29	0.78
▨ Option II	0.04	0.35	0.88
□ Option III	0.09	0.48	1.28

Fig. 7 Computational times for the manycore 3D-DWT algorithm. GOP size: 64. GTX280.

In most cases the horizontal and/or vertical dimension is greater than the maximum number of threads per block, so, each thread must compute more than one coefficient. On the other hand, the considered GOP size is always lower than the maximum number of threads per block, therefore in this case each thread computes only one coefficient. Hence, the number of threads per block in the 1D-DWT computation depends on the GOP size considered. In order to compute the 2D-DWT we have run several tests with different number of threads per block, from 16 up to the maximum (512). After analyzing test results, performance results do not change with values ranging from 64 to 256 threads per block. We can conclude the results are not affected using values from 64 to 256.

The platform used to test the manycore algorithm is the NVIDIA GTX 280 GPU that contains 30 multiprocessors with 8 cores in each multiprocessor, 1 GB of global memory and 16 KB of shared memory by block (or SM), this device is available on the multicore Intel Core 2 Quad Q6600 2.4 GHz.

Fig. 7 presents results for several grayscale video frame resolutions and a GOP size equal to 64 on the GTX280 platform. We can observe that option I is the optimal choice, obtaining in some cases a significant improvement. All results presented for the manycore 3D wavelet transform do not include the communication times between host and GPU. Note that the 3D-WDT is the first step of a more complex process, such as a video codec.

In order to clarify the optimal CUDA algorithm developed, we want to remark that the GOP video sequence is stored in the GPU global memory, and the working memory (that stores the current data row, data column or array in the temporal axis) is stored in the GPU shared memory (see Fig. 8). On the other hand, we are translating the strategy of the multicore algorithm to the manycore algorithm. As we have said, we have tested three options to develop de 3D-DWT CUDA algorithm, and the best results are obtained following the option I (see Fig. 7). We have developed three basic kernels in the optimal algorithm. The first one computes the first stage of the 2D-DWT, the second one computes the second stage of the 2D-DWT and the third one computes the 1D-DWT in the temporal axis. As we can see in Fig. 8, each block of threads

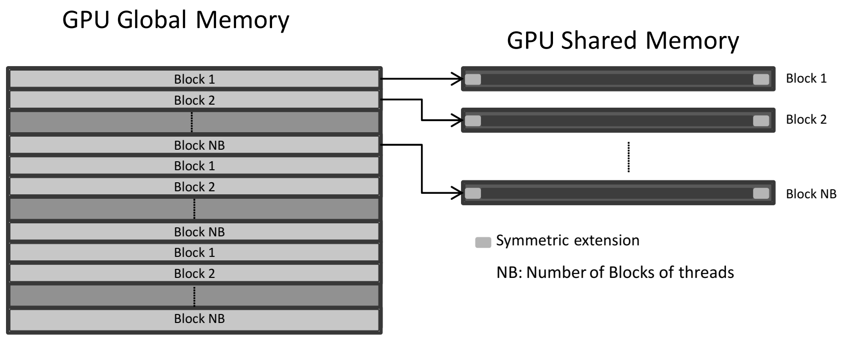


Fig. 8 Parallel structure for the GPU algorithm.

computes the components associated to one row, if the number of blocks is less than the column dimension each block must to compute more than one row. Note that, the computation is performed over the working memory (i.e. the shared memory), and then all threads of one block must to cooperate in order to copy the current data row (or column or array in temporal axis) from global memory to shared memory. We can extend the row computation structure showed in Fig. 8, to the computation of both the column computation and the array in the temporal axis computation.

As we have said, the first stage in the kernels computations is to move the data from global memory to shared memory. Best results are obtained when each thread moves pixels with stride equal to the number of threads by block in the kernel (see Fig. 9), if the working data dimension is greater than the number of threads by block. Moreover, we must perform the symmetric extension on both borders, being performed by the lower rank threads in the first border and by the higher rank threads in the second border.

5 Performance Evaluation

In this section we discuss the behavior of the parallel algorithm described in previous sections and we compare it against a recent optimized multicore proposal presented in [18]. Fig. 10 presents the 3D-DWT computational times and their associated efficiencies for a video frame resolution of 1280×640 varying the GOP size and the number of processes. Fig. 10(a) shows the good computation behavior of the parallel algorithm. In the 3D-DWT there is an intensive use of the memory, therefore the improvement in the use of the cache memory and data locality justifies the efficiencies greater than 1 showed in Fig. 10(b). Efficiency values correspond to executions on the multicore Q6600 platform. However, in Fig. 11(b) this fact is not observed for the multicore HP Proliant SL390 due to the higher memory access performance respect to the multicore Q6600. The HP Proliant SL390 architecture provides a high-bandwidth memory access, through the Intel QPI Speed 64GT/s, therefore, the global performance improvement is less significant than in the Q6600 platform. In Fig. 11

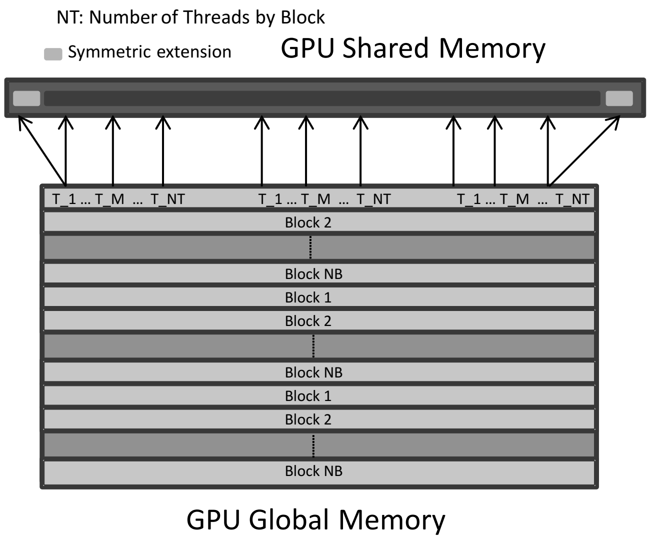
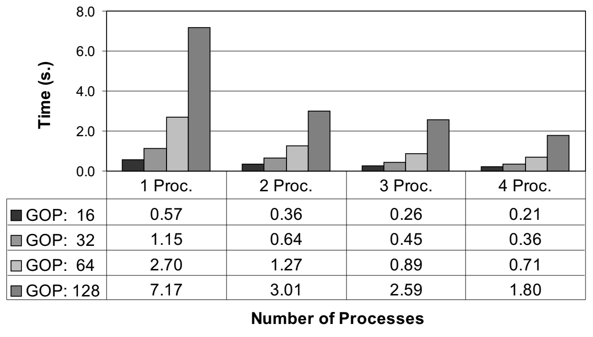


Fig. 9 Memory copy from global memory to shared memory.

we also present the computational times and their associated efficiencies for the multicore HP Proliant SL390. The efficiencies obtained on both platforms are similar, however, comparing data obtained from video frames of different resolutions we can conclude that the behavior on the multicore Q6600 becomes worse than on the multicore HP Proliant SL390, as the GOP size increases, i.e. when the global memory size increases. Note that the data presented in figures 10(b) and 11(b) correspond to different video frame resolutions.

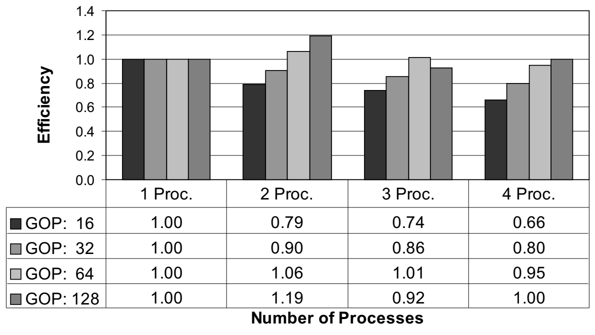
The GOP size is an important parameter in the 3D-DWT computation, when applied to video coding, because the average video quality increases as we increase the GOP size due to the minor GOP boundary effect. However, the computational load and memory requirements increase. Ideally, the GOP size would be equal to the total number of video frames, since this is not possible due to the device memory restrictions, we must select the GOP size attending to both the video quality and the computational time. As we can see in figures 10(a) and 11(a) the computational time increases as the GOP size increases. The minimum GOP size in our algorithm is 16 due to the four wavelet decomposition levels performed in the 3D-DWT (2^4). In Fig. 12 we show the percentage of increment in computational time, respect to the computational time with GOP size equal to 16. Regarding results presented in figures 10 and 11, we can conclude that the optimal value for the GOP size is equal to 64 or 128, being the final decision independent of the platform and probably we must set this parameter by considering if the memory requirements is a critical parameter or not.

Note that the number of frames (or pictures) computed is the GOP size value. Therefore when the GOP size increases, the border effects in the temporal axis decrease, and, obviously the computational time increases due to the



	1 Proc.	2 Proc.	3 Proc.	4 Proc.
■ GOP: 16	0.57	0.36	0.26	0.21
■ GOP: 32	1.15	0.64	0.45	0.36
□ GOP: 64	2.70	1.27	0.89	0.71
■ GOP: 128	7.17	3.01	2.59	1.80

(a) Time (s.).



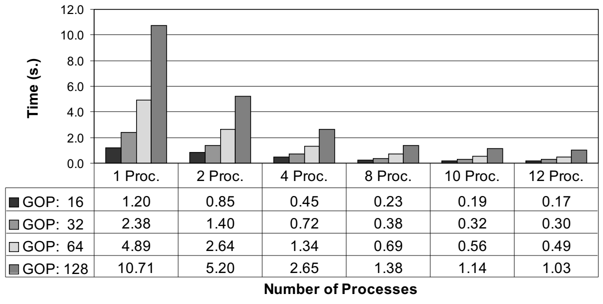
	1 Proc.	2 Proc.	3 Proc.	4 Proc.
■ GOP: 16	1.00	0.79	0.74	0.66
■ GOP: 32	1.00	0.90	0.86	0.80
□ GOP: 64	1.00	1.06	1.01	0.95
■ GOP: 128	1.00	1.19	0.92	1.00

(b) Efficiency.

Fig. 10 3D wavelet algorithm. Compiler: GCC. Compiler flags: -O3 -openmp. Frame size: 1280×640. Multicore Q6600.

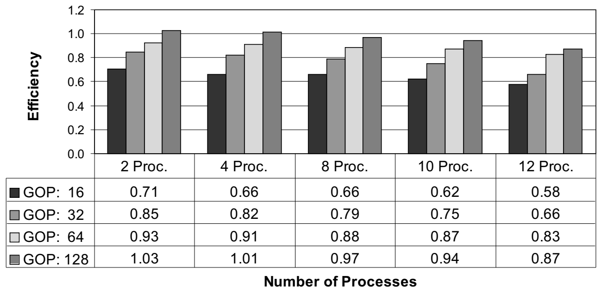
larger number of frames to be computed. In Fig. 13 we present the computational time per frame corresponding to Fig. 12 data. We can observe that the parallel algorithm improves its behavior when both the number of processes and the GOP size increase. We want to remark that setting the GOP size equal to 256, for medium and high resolution video frames, the results obtained are not good due to the global memory size requirement. Setting the GOP size equal to 128 reduces the border effects while setting the GOP size equal to 64 reduces the memory requirements. Both GOP size values obtain the best results, as it can be seen in Fig. 13, in terms of computation times per frame.

We have presented an exhaustive analysis of our parallel algorithm, showing its behavior according to the possible modifications of all parameters. As we have seen, the parallel algorithm obtains good efficiency, with the proper parameters setting, using the available cores, up to 12 in the multicore HP Proliant SL390 and up to 4 in the multicore Q6600. At this time we will perform a comparative analysis against the recent 3D-DWT algorithm presented in [18], which presents some interesting optimization techniques. Both algorithms under comparison use different methods to compute the 3D-DWT, in



	1 Proc.	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ GOP: 16	1.20	0.85	0.45	0.23	0.19	0.17
■ GOP: 32	2.38	1.40	0.72	0.38	0.32	0.30
□ GOP: 64	4.89	2.64	1.34	0.69	0.56	0.49
■ GOP: 128	10.71	5.20	2.65	1.38	1.14	1.03

(a) Time (s.).



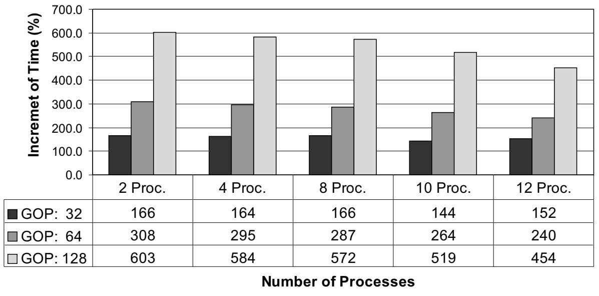
	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ GOP: 16	0.71	0.66	0.66	0.62	0.58
■ GOP: 32	0.85	0.82	0.79	0.75	0.66
□ GOP: 64	0.93	0.91	0.88	0.87	0.83
■ GOP: 128	1.03	1.01	0.97	0.94	0.87

(b) Efficiency.

Fig. 11 3D wavelet algorithm. Compiler: ICC. Compiler flags: -fast -openmp. Frame size: 1920 × 1024. Multicore HP Proliant SL390.

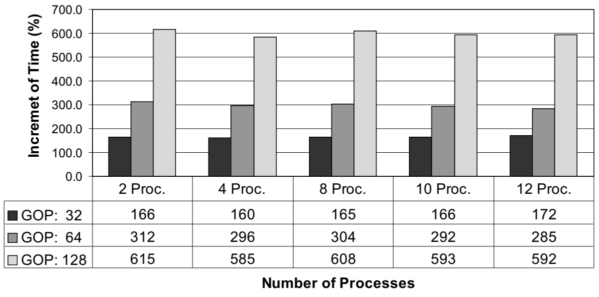
particular the reference algorithm uses the Daubechies W_4 filter instead of the Daubechies 9/7 filter used in our algorithm. It should be noted that our algorithm performs four decomposition levels to compute the 3D-DWT, while the reference algorithm presented in [18] performs only one decomposition level. Therefore, the optimization techniques used in both algorithms can not be the same. Furthermore we use the symmetric extension technique to avoid boundary effects, while the reference algorithm does not apply any specific technique for this purpose.

The platforms used to test both algorithms are very similar. Our platform has an Intel Q6600 quad-core processor and the reference algorithm has been run on an Intel Q6700 quad-core processor, i.e. the reference algorithm has been tested on a platform with a slightly higher performance. Fig. 14 shows the computational times to compute the 3D-DWT for several video frame sizes, using 4 processes and a GOP size equal to 64. The results provided of the reference algorithm depend substantially on the compiler, as we can see in Fig. 14(b), while our algorithm shows a lower compiler dependency, as it showed in Fig. 6. The results shown in Fig. 14(a) are obtained with the GCC compiler, because our multicore Q6600 does not provide the ICC compiler. The computational times presented in Fig. 14 are obtained for different video



	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ GOP: 32	166	164	166	144	152
▨ GOP: 64	308	295	287	264	240
□ GOP: 128	603	584	572	519	454

(a) Frame size: 1280 × 640.



	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ GOP: 32	166	160	165	166	172
▨ GOP: 64	312	296	304	292	285
□ GOP: 128	615	585	608	593	592

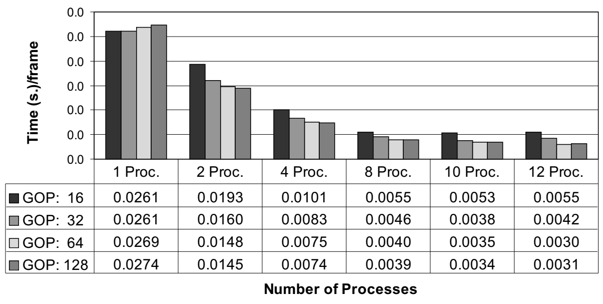
(b) Frame size: 1920 × 1024.

Fig. 12 Increment of time (%) respect to the GOP size equal to 16. Compiler: ICC. Compiler flags: -fast -openmp. Multicore HP Proliant SL390.

frame resolutions, because we work with standard video frame resolutions and the reference algorithm works with square video frame resolutions.

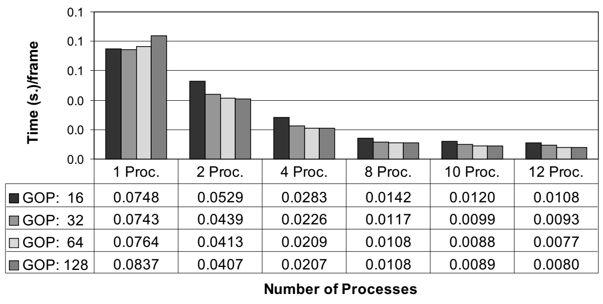
In Fig. 15 we analyze the algorithm throughput as the number of megapixels per second in both algorithms. We can conclude that our algorithm shows a greater performance degradation as the video frame resolution increases. This is due to the previously discussed differences, both the number of wavelet decomposition levels and the symmetric extension performed in our algorithm. Techniques used to improve the reference algorithm do not improve the performance of our algorithm. Note that we use an extra amount of memory to store the working data as showed in Table 1. We want to remark that our algorithm avoids the use of twice the video size to store the computed coefficients through this working memory.

As mentioned both algorithms use different filters in order to compute the 3D-DWT, which means that the computational load per pixel differs on both algorithms. Therefore in Fig. 16 we present results in terms of GFLOPS. We have computed the GFLOPS of the experiments reported in [18] considering the video frame resolution, the GOP size and the Daubechies W_4 filter. Attending to the results obtained using the same compiler, our algorithm is able to compute up to 4 times more GFLOPS than the reference algorithm. Ana-



	1 Proc.	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ GOP: 16	0.0261	0.0193	0.0101	0.0055	0.0053	0.0055
▨ GOP: 32	0.0261	0.0160	0.0083	0.0046	0.0038	0.0042
□ GOP: 64	0.0269	0.0148	0.0075	0.0040	0.0035	0.0030
▨ GOP: 128	0.0274	0.0145	0.0074	0.0039	0.0034	0.0031

(a) Frame size: 1280 × 640.



	1 Proc.	2 Proc.	4 Proc.	8 Proc.	10 Proc.	12 Proc.
■ GOP: 16	0.0748	0.0529	0.0283	0.0142	0.0120	0.0108
▨ GOP: 32	0.0743	0.0439	0.0226	0.0117	0.0099	0.0093
□ GOP: 64	0.0764	0.0413	0.0209	0.0108	0.0088	0.0077
▨ GOP: 128	0.0837	0.0407	0.0207	0.0108	0.0089	0.0080

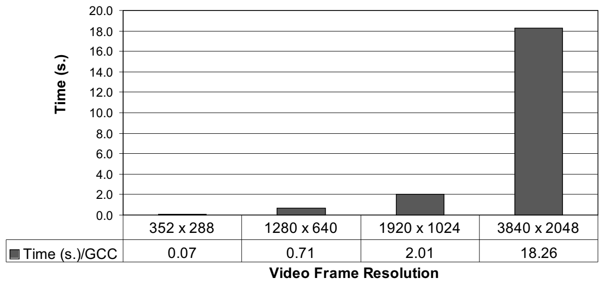
(b) Frame size: 1920 × 1024.

Fig. 13 Computational time per frame. Compiler: ICC. Compiler flags: -fast -openmp. Multicore HP Proliant SL390.

lyzing different compilers, our algorithm increases the GFLOPS using the free compiler GCC respect to the corporate ICC compiler.

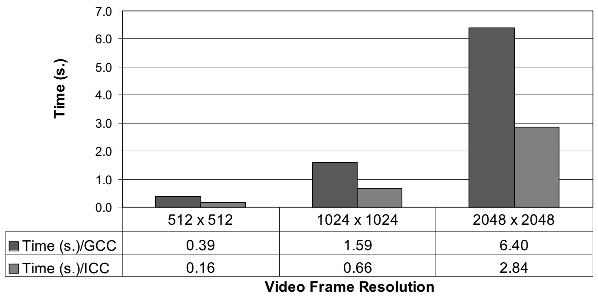
Finally we analyze the manycore algorithm. Fig. 17 presents the speed-up for various grayscale video frame resolutions and GOP sizes on the GTX280 platform. The speed-up is computed with respect to the computational times obtained using one core of the platform Q6600 platforms plus GTX280. The speed-ups obtained for medium and high resolution images allows us to apply the CUDA 3D-DWT algorithm to improve the performance of a video coder based on the 3D-DWT. Missing data in Fig. 17 corresponds to the case where the required global memory is greater than the memory available in the GTX280 ($1GB$). This fact confirms the importance of the in-place computation of our algorithm.

It should be noted that the average GFLOPS obtained in the manycore GTX280, for medium and high resolution images is around 4 GFLOPS, obtaining a significant improvement respect the data shown in Fig. 17 using the optimal number of threads.



	352 x 288	1280 x 640	1920 x 1024	3840 x 2048
■ Time (s.)/GCC	0.07	0.71	2.01	18.26

(a) Developed algorithm. Multicore Q6600.



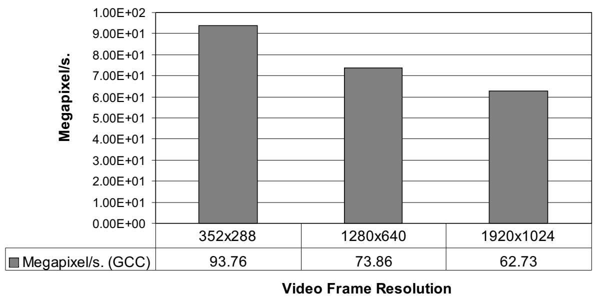
	512 x 512	1024 x 1024	2048 x 2048
■ Time (s.)/GCC	0.39	1.59	6.40
▨ Time (s.)/ICC	0.16	0.66	2.84

(b) Reference algorithm. Multicore Q6700.

Fig. 14 Computational times for 3D wavelet algorithm. GOP size: 64. Number of processes: 4.

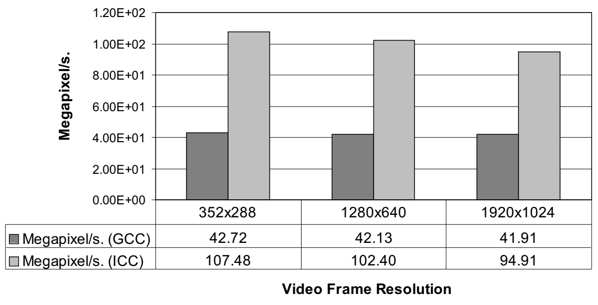
6 Conclusions

We have presented the multicore-based algorithm developed using the OpenMP paradigm, that performs the 3D discrete wavelet transform and the manycore-based algorithm developed using CUDA. We have analyzed the behavior of the proposed algorithms when running on two different shared-memory platforms and on the GPU GTX280. Furthermore, we have compared our algorithm against a recent multicore algorithm proposed in [18]. The multicore-based algorithm obtains speed-ups closely ideal depending on the video frame resolution and the GOP size, running on a relatively low computing power platform as the Q6600 multicore platform, when compared to the sequential CPU algorithm. When running on the HP Proliant SL390 G7 platform, our algorithm obtains good efficiencies even using the maximum number of available cores, depending on the video frame resolution and the GOP size. In this case the efficiencies achieved are greater than 83%. Furthermore, we do not require twice the video size to compute the 3D-DWT and the increased memory, even up to 12 processes, is negligible. This characteristic is especially important when considering the manycore algorithm, and moreover the manycore algorithm obtains significant improvements respect to the multicore algorithm one.



	352x288	1280x640	1920x1024
■ Megapixel/s. (GCC)	93.76	73.86	62.73

(a) Developed algorithm. Multicore Q6600.



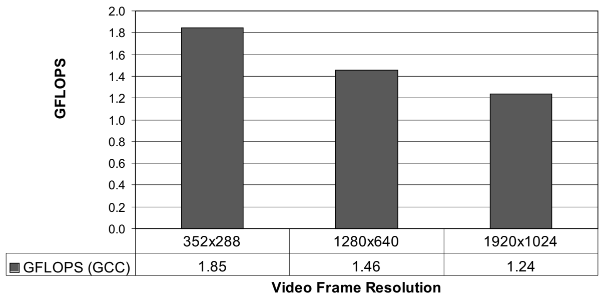
	352x288	1280x640	1920x1024
■ Megapixel/s. (GCC)	42.72	42.13	41.91
□ Megapixel/s. (ICC)	107.48	102.40	94.91

(b) Reference algorithm. Multicore Q6700.

Fig. 15 Megapixels per second for 3D wavelet algorithm. GOP size: 64. Number of processes: 4.

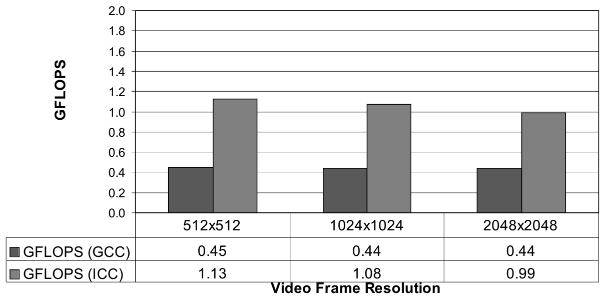
References

1. P. Campisi and A. Neri. Video watermarking in the 3D-DWT domain using perceptual masking. In *IEEE International Conference on Image Processing*, pages 997–1000, September 2005.
2. P. Schelkens, A. Munteanu, J. Barbariend, M. Galca, X. Giro-Nieto, and J. Cornelis. Wavelet coding of volumetric medical datasets. *IEEE Transactions on Medical Imaging*, 22(3):441–458, March 2003.
3. P.L. Dragotti and G. Poggi. Compression of multispectral images by three-dimensional SPITH algorithm. *IEEE Transactions on Geoscience and Remote Sensing*, 38(1):416–428, January 2000.
4. M. Aviles, F. Moran, and N. Garcia. Progressive lower trees of wavelet coefficients: Efficient spatial and SNR scalable coding of 3D models. *Lecture Notes in Computer Science*, 3767:61–72, 2005.
5. C.I Podilchuk, N.S. Jayant, and N. Farvardin. Three dimensional subband coding of video. *IEEE Tran. on Image Processing*, 4(2):125–135, February 1995.
6. D. Taubman and A. Zakhor. Multirate 3-D subband coding of video. *IEEE Tran. on Image Processing*, 3(5):572–588, September 1994.
7. J.M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12), December 1993.
8. A. Said and A. Pearlman. A new, fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits, Systems and Video Technology*, 6(3):243–250, 1996.



	352x288	1280x640	1920x1024
▨ GFLOPS (GCC)	1.85	1.46	1.24

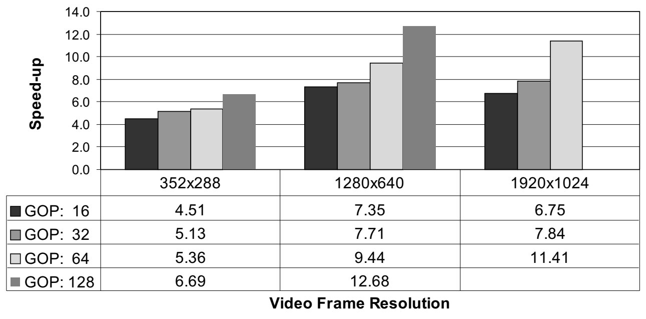
(a) Developed algorithm. Multicore Q6600.



	512x512	1024x1024	2048x2048
▨ GFLOPS (GCC)	0.45	0.44	0.44
▨ GFLOPS (ICC)	1.13	1.08	0.99

(b) Reference algorithm. Multicore Q6700.

Fig. 16 GFLOPS for 3D wavelet algorithm. GOP size: 64. Number of processes: 4.



	352x288	1280x640	1920x1024
■ GOP: 16	4.51	7.35	6.75
▨ GOP: 32	5.13	7.71	7.84
□ GOP: 64	5.36	9.44	11.41
▨ GOP: 128	6.69	12.68	

Fig. 17 Speed-up for the manycore 3D-DWT algorithm. GTX280.

9. J. Oliver and M. P. Malumbres. Low-complexity multiresolution image compression using wavelet lower trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(11):1437–1444, 2006.
10. Y. Chen and W.A. Pearlman. Three-dimensional subband coding of video using the zero-tree method. In *Visual Communications and Image Processing*, volume Proc. SPIE 2727, pages 1302–1309, March 1996.
11. J. Luo, X. Wang, C.W. Chen, and K.J. Parker. Volumetric medical image compression with three-dimensional wavelet transform and octave zerotree coding. In *Visual Communications and Image Processing*, volume Proc. SPIE 2727, pages 579–590, March

1996.
12. B.J. Kim, Z. Xiong, and W.A. Pearlman. Low bit-rate scalable video coding with 3D set partitioning in hierarchical trees (3D SPIHT). *IEEE Transactions on Circuits and Systems for Video Technology*, 10:1374–1387, December 2000.
13. O. Lopez, M. Martinez-Rach, P. Piñol, M.P. Malumbres, and J.Oliver. Lower bit-rate video coding with 3D lower trees (3D-LTW). In *IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pages 3105–3108, 1998.
14. T.-T. Wong, C.-S. Leung, P.-A. Heng, and J. Wang. Discrete wavelet transform on consumer-level graphics hardware. *Multimedia, IEEE Transactions on*, 9(3):668 –673, april 2007.
15. C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, and F. Tirado. Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting. *Parallel and Distributed Systems, IEEE Transactions on*, 19(3):299 –310, march 2008.
16. J. Franco, G. Bernabé, J. Fernández, M.E. Acacio, and M. Ujaldón. The GPU on the 2D wavelet transform. survey and contributions. In *In proceedings of Para 2010: State of the Art in Scientific and Parallel Computing*, 2010.
17. V. Galiano, O. López, M.P. Malumbres, and H. Migallón. Improving the discrete wavelet transform computation from multicore to gpu-based algorithms. In *In proceedings of International Conference on Computational and Mathematical Methods in Science and Engineering*, 2011.
18. J. Franco, G. Bernabé, J. Fernández, and M. Ujaldón. Parallel 3D fast wavelet transform on manycore gpus and multicore cpus. *Procedia Computer Science*, 1(1):1101 – 1110, 2010.
19. S. G. Mallat. A theory for multi-resolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.
20. OpenMP application program interface, version 3.1. *OpenMP Architecture Review Board. `http://www.openmp.org`*, 2011.
21. ICC, intel software network. *`http://software.intel.com/en-us/intel-compilers/`*, 2009-2011.
22. GCC, the GNU compiler collection. *Free Software Foundation, Inc. `http://gcc.gnu.org`*, 2009-2012.
23. J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. In *Queue*, volume 6, pages 40–53, 2008.
24. NVIDIA Corporation. Nvidia CUDA C programming guide. version 3.2.